Network Working Group
Request for Comments: 993
Obsoletes: RFC-984

David D. Clark (MIT)
Mark L. Lambert (MIT)
December 1986

PCMAIL: A Distributed Mail System for Personal Computers

1. Status of this Document

This document is a discussion of the Pcmail workstation-based distributed mail system. It is a revision of the design published in NIC RFC-984. The revision is based on discussion and comment from a variety of sources, as well as further research into the design of interactive Pcmail clients and the use of client code on machines other than IBM PCs. As this design may change, implementation of this document is not advised. Distribution of this memo is unlimited.

2. Introduction

Pcmail is a distributed mail system providing mail service to an arbitrary number of users, each of whom owns one or more workstations. Pcmail's motivation is to provide very flexible mail service to a wide variety of different workstations, ranging in power from small, resource-limited machines like IBM PCs to resource-rich (where "resources" are primarily processor speed and disk space) machines like Suns or Microvaxes. It attempts to provide limited service to resource-limited workstations while still providing full service to resource-rich machines. It is intended to work well with machines only infrequently connected to a network as well as machines permanently connected to a network. It is also designed to offer diskless workstations full mail service.

The system is divided into two halves. The first consists of a single entity called the "repository". The repository is a storage center for incoming mail. Mail for a Pcmail user can arrive externally from the Internet or internally from other repository users. The repository also maintains a stable copy of each user's mail state (this will hereafter be referred to as the user's "global mail state"). The repository is therefore typically a computer with a large amount of disk storage.

The second half of Pcmail consists of one or more "clients". Each Pcmail user may have an arbitrary number of clients, typically single-user workstations. The clients provide a user with a friendly means of accessing the user's global mail state over a network. In order to make the interaction between the repository and a user's clients more efficient, each client maintains a local copy of its

user's global mail state, called the "local mail state". It is assumed that clients, possibly being small personal computers, may not always have access to a network (and therefore to the global mail state in the repository). This means that the local and global mail states may not be identical all the time, making synchronization between local and global mail states necessary.

Clients communicate with the repository via the Distributed Mail System Protocol (DMSP); the specification for this protocol appears in appendix A. The repository is therefore a DMSP server in addition to a mail end-site and storage facility. DMSP provides a complete set of mail manipulation operations ("send a message", "delete a message", "print a message", etc.). DMSP also provides special operations to allow easy synchronization between a user's global mail state and his clients' local mail states. Particular attention has been paid to the way in which DMSP operations act on a user's mail state. All DMSP operations are failure-atomic (that is, they are guaranteed either to succeed completely, or leave the user's mail state unchanged). A client can be abruptly disconnected from the repository without leaving inconsistent or damaged mail states.

Pcmail's design has been directed by the characteristics of currently available workstations. Some workstations are fairly portable, and can be packed up and moved in the back seat of an automobile. A few are truly portable--about the size of a briefcase--and battery-powered. Some workstations have constant access to a high-speed local-area network; pcmail should allow for "on-line" mail delivery for these machines while at the same time providing "batch" mail delivery for other workstations that are not always connected to a network. Portable and semi-portable workstations tend to be resource-poor. A typical IBM PC has a small amount (typically less than one megabyte) of main memory and little in the way of mass storage (floppy-disk drives that can access perhaps 360 kilobytes of data). Pcmail must be able to provide machines like this with adequate mail service without hampering its performance on more resource-rich workstations. Finally, all workstations have some common characteristics that Pcmail should take advantage of. For instance, workstations are fairly inexpensive compared to the various time-shared systems that most people use for mail service. This means that people may own more than one workstation, perhaps putting a Microvax in an office and an IBM PC at home.

Pcmail's design reflects the differing characteristics of the various workstations. Since one person can own several workstations, Pcmail allows users multiple access points to their mail state. Each Pcmail user can have several client workstations, each of which can access the user's mail by communicating with the repository over a network. The clients all maintain local copies of the user's global mail state, and synchronize the local and global states using DMSP.

It is also possible that some workstations will only infrequently be

connected to a network (and thus be able to communicate with the repository). The Pcmail design therefore allows two modes of communication between repository and client. "Interactive mode" is used when the client is always connected to the network. Any changes to the client's local mail state are immediately also made to the repository's global mail state, and any incoming mail is immediately transmitted from repository to client. "Batch mode" is used by clients that have infrequent access to the repository. Users manipulate the client's local mail state, queueing the changes locally. When the client is next connected to the repository, the changes are executed, and the client's local mail state is synchronized with the repository's global mail state.

Finally, the Pcmail design minimizes the effect of using a resource-poor workstation as a client. Mail messages are split into two parts: a "descriptor" and a "body". The descriptor is a capsule message summary whose length (typically about 100 bytes) is independent of the actual message length. The body is the actual message text, including an RFC-822 standard message header. While the client may not have enough storage to hold a complete set of messages, it can usually hold a complete set of descriptors, thus providing the user with at least a summary of his mail state. For clients with extremely limited resources, Pcmail allows the storage of partial sets of descriptors. Although this means the user does not have a complete local mail state, he can at least look at summaries of some messages. In the cases where the client cannot immediately store message bodies, it can always pull them over from the repository as storage becomes available.

The remainder of this document is broken up into sections discussing the following:

- The repository architecture
- DMSP, its operations, and motivation for its design
- The client architecture
- A typical DMSP session between the repository and a client
- The current Pcmail implementation

3. Repository architecture

A typical machine running repository code has a relatively powerful processor and a large amount of disk storage. It must also be a permanent network site, for two reasons. First clients communicate with the repository over a network, and rely on the repository's being available at any time. Second, people sending mail to repository users rely on the repository's being available to receive mail at any

time.

The repository must perform several tasks. First, and most importantly, the repository must efficiently manage a potentially large number of users and their mail states. Mail must be reliably stored in a manner that makes it easy for multiple clients to access the global mail state and synchronize their local mail states with the global state. Since a large category of electronic mail is represented by bulletin boards (bboards), the repository should efficiently manage bboard mail, using a minimum of storage to store bboard messages in a manner that still allows any user subscribing to the bboard to read the mail. Second, the repository must be able to communicate efficiently with its clients. The protocol used to communicate between repository and client must be reliable and must provide operations that (1) allow typical mail manipulation, and (2) support Pcmail's distributed nature by allowing efficient synchronization between local and global mail states. Third, the repository must be able to process mail from sources outside the repository's own user community (a primary outside source is the Internet). Internet mail will arrive with a NIC RFC-822 standard message header; the recipient names in the message must be properly translated from the RFC-822 namespace into the repository's namespace.

3.1. Management of user mail state

Pcmail divides the world into a community of users. Each user is referred to by a user object. A user object consists of a unique name, a password (which the user's clients use to authenticate themselves to the repository before manipulating a global mail state), a list of "client objects" describing those clients belonging to the user, and a list of "mailbox objects".

A client object consists of a unique name and a status. A user has one client object for every client he owns; a client cannot communicate with the repository unless it has a corresponding client object in a user's client list. Client objects therefore serve as a means of identifying valid clients to the repository. Client objects also allow the repository to manage local and global mail state synchronization; the repository associates with every global state change a list of client objects corresponding to those clients which have not recorded the global change locally.

A client's status is either "active" or "inactive". The repository defines inactive clients as those clients which have not connected to the repository within a set time period (one week in the current repository implementation). When an inactive client does connect to the repository, the repository notifies the client that it has been "reset". The repository resets a client by marking all messages in the user's mail state as having changed since the client last logged in. When the client next synchronizes with the repository, it will receive a complete copy of the repository's global mail state. A

forced reset is performed on the assumption that enough global state changes occur in a week that the client would spend too much time performing an ordinary local state-global state synchronization.

Messages are stored in mailboxes. Users can have an arbitrary number of mailboxes, which serve both to store and to categorize messages. A mailbox object both names a mailbox and describes its contents. Mailboxes are identified by a unique name; their contents are described by three numeric values. The first is the total number of messages in the mailbox, the second is the total number of unseen messages (messages that have never been seen by the user via any client) in the mailbox, and the third is the mailbox's next available message unique identifier (UID). The above information is stored in the mailbox object to allow clients to get a summary of a mailbox's contents without having to read all the messages within the mailbox.

Some mailboxes are special, in that other users may read the messages stored in them. These mailboxes are called "bulletin board mailboxes" or "bboard mailboxes". The repository uses bboard mailboxes to store bboard mail. Bboard mailboxes differ from ordinary mailboxes in the following ways:

- Their names are unique across the entire repository; for instance, only one bboard mailbox named "sf-lovers" may exist in the entire repository community. This does not preclude other users from having an ordinary mailbox named "sf-lovers".

- Subscribers to the bboard are granted read-only access to the messages in the bboard mailbox. The bboard mailbox's owner (typically the system manager) has read/update/delete access to the mailbox.

A bboard subscriber keeps track of the messages he has looked at via a bboard object. The bboard object contains the name of the bboard, its owner (the user who owns the bboard mailbox where all the messages are stored), and the UID of the first message not yet seen by the subscriber .

Users gain read-only access to a bboard by "subscribing" to it; they lose that access when they "unsubscribe" to it.

Associated with each mailbox are an arbitrary number of message objects. Each message is broken into two parts--a "descriptor", which contains a summary of useful information about the message, and a "body", which is the message text itself, including its NIC RFC-822 message header. Each message is assigned a monotonically increasing UID based on the owning mailbox's next available UID. Each mailbox has its own set of UIDs which, together with the mailbox name and user name, uniquely identify the message within the repository.

A descriptor holds the following information: the message UID, the message size in bytes and lines, four "useful" message header fields (the "date:", "to:", "from:", and "subject:" fields), and sixteen flags. These flags are given identifying numbers 0 through: 15. Eight of these flags are reserved for the repository's use. Some of these are actually used by the repository, while others are simply held for informational purposes. In the current repository implementation these flags mark:

- (#0) whether it has been deleted
- (#1) whether the message has been seen
- (#2) whether it has been forwarded to the user
- (#3) whether it has been forwarded by the user
- (#4) whether it has been filed (written to a text file outside the repository)
- (#5) whether it has been printed (locally or remotely)
- (#6) whether it has been replied to
- (#7) whether it has been copied to another mailbox

The remaining eight flags are reserved for future use.

Descriptors serve as an efficient means for clients to get message information without having to waste time retrieving the message from the repository.

3.2. Repository-to-RFC-822 name translation

"Address objects" provide the repository with a means for translating the RFC-822-style mail addresses in Internet messages into repository names. The repository provides its own namespace for message identification. Any message is uniquely identified by the triple (user-name, mailbox-name, message-UID). Any mailbox is uniquely identified by the pair (user-name, mailbox-name). Thus to send a message between two repository users, a user would address the message to (user-name, mailbox-name). The repository would deliver the message to the named user and mailbox, and assign it a UID based on the requested mailbox's next available UID.

In order to translate between RFC-822-style mail addresses and repository names, the repository maintains a list of address objects. Each address object is an association between an RFC-822-style address and a (user-name, mailbox-name) pair. When mail arrives from the Internet, the repository can use the address object list to translate the recipients into (user-name, mailbox-name) pairs and

route the message correctly.

4. Communication between repository and client: DMSP

The Distributed Mail System Protocol (DMSP) is a block-stream protocol that defines and manipulates the objects mentioned in the previous section. It has been designed to work with Pcmail's single-repository/multiple-client model of the world. In addition to providing typical mail manipulation functions, DMSP provides functions that allow easy synchronization of global and local mail states.

DMSP is implemented on top of the Unified Stream Protocol (USP), specified in MIT-LCS RFC-272. USP provides a reliable virtual circuit block-stream connection between two machines. It defines a basic set of data types ("strings", "integers", "booleans", etc.); instances of these data types are grouped in an application-defined order to form USP blocks. Each USP block is defined by a numeric "block type"; a USP application can thus interpret a block's contents based on knowledge of the block's type. DMSP consists of a set of operations, each of which is comprised of one or more different USP blocks that are sent between repository and client.

A DMSP session proceeds as follows: a client begins the session with the repository by opening a USP connection to the repository's machine. The client then authenticates both itself and its user to the repository with a "login" operation. If the authentication is successful, the user performs an arbitrary number of DMSP operations before ending the session with a "logout" operation (at which time the connection is closed by the repository).

Because DMSP can manipulate a pair of mail states (local and global) at once, it is extremely important that all DMSP operations are failure-atomic. Failure of any DMSP operation must leave both states in a consistent, known state. For this reason, a DMSP operation is defined to have failed unless an explicit acknowledgement is received by the operation initiator. This acknowledgement can take one of two basic forms, based on two broad categories that all DMSP operations fall into. First, an operation can be a request to perform some mail state modification, in which case the repository will acknowledge the request with either an "ok" or a "failure" (in which case the reason for the failure is also returned). Second, an operation can be a request for information, in which case the request is acknowledged by the repository's providing the information to the client. Operations such as "delete a message" fall into the first category; operations like "send a list of mailboxes" fall into the second category.

Following is a general discussion of all the DMSP operations. The operations are broken down by type: general operations, user operations, client operations, mailbox operations, address operations, bboard operations, and message operations.

4.1. General operations

The first group of DMSP operations perform general functions that operate on no one particular class of object. DMSP has two general operations, which provide the following services:

In order to prevent protocol version skew between clients and the repository, DMSP provides a "send-version" operation. The client supplies its DMSP version number as an argument; the operation succeeds if the supplied version number matches the repository's DMSP version number. It fails if the two version numbers do not match. The version number is an unsigned quantity, like "100", "101", "200". The "send-version" operation should be the first that a client sends to the repository, since no other operation my work if the client and repository are using different versions of DMSP.

Users can send mail to other users via the "send-message" operation. The message must have an Internet-style header as defined by NIC RFC-822. The repository takes the message and distributes it to the mailboxes specified on the "to:", "cc:", and "bcc:" fields of the message header. If one or more of the mailboxes exists outside the repository's user community, the repository is responsible for handing the message to a local SMTP server.

An OK is sent from the repository only if the entire message was successfully transmitted. If the message was destined for the Internet, the send-message operation is successful if the message was successfully transmitted to the local SMTP server.

4.2. User operations

The next series of DMSP operations manipulates user objects. The most common of these operations are "login" and "logout". A client must perform a login operation before being able to access a user's mail state. A DMSP login block contains five items: (1) the user's name, (2) the user's password, (3) the name of the client performing the login, (4) a flag telling the repository to create a client object for the client if one does not exist, and (5) a flag set to TRUE if the client wishes to operate in "batch mode" and FALSE if the client wishes to operate in "interactive" mode. The flag value allows the repository to tune internal parameters for either mode of operation.

The repository can return either an OK block (indicating successful authentication), a FAILURE block (indicating failed authentication), or a FORCE-RESET block. This last is sent if the client logging in has been marked as "inactive" by the repository (clients are marked inactive if they have not connected to the repository in over a week). The FORCE-RESET block indicates that the client should erase its local mail state and pull over a complete version of the repository's mail state. This is done on the assumption that so many

mail state changes have been made in a week that it would be inefficient to perform a normal synchronization.

When a client has completed a session with the repository, it performs a logout operation. This allows the repository to perform any necessary cleanup before closing the USP connection.

A user can change his password via the "set-password" operation. The operation works much the same as the UNIX change-password operation, taking as arguments the user's current password and a desired new password. If the current password given matches the user's current password, the user's current password is changed to the new password given.

4.3. Client operations

DMSP provides four operations to manipulate client objects. The first, "list-clients", tells the repository to send the user's client list to the requesting client. The list takes the form of a series of (client-name, status) pairs. The status is either 0 (inactive) or 1 (active).

The "create-client" operation allows a user to add a client object to his list of client objects. Although the login operation duplicates this functionality via the "create-this-client?" flag, the add-client operation is a useful means of creating a number of new client objects while logged into the repository via an existing client. The create-client operation requires the name of the client to add.

The "delete-client" operation removes an existing client object from a user's client list. The client being removed cannot be in use by anyone at the time.

The last client operation, "reset-client", causes the repository to mark all messages in the user's mail state as having changed since the client last logged in. When a client next synchronizes with the repository, it will end up receiving a complete copy of the repository's global mail state. This is useful for two reasons. First, a client's local mail state could easily become lost or damaged, especially if it is stored on a floppy disk. Second, if a client has been marked as inactive by the repository, the reset-client operation provides a fast way of resynchronizing with the repository, assuming that so many differences exist between the local and global mail states that a normal synchronization would take far too much time.

4.4. Mailbox operations

DMSP supports five operations that manipulate mailbox objects. First, "list-mailboxes" has the repository send to the requesting client information on each mailbox. This information consists of the

mailbox name, total message count, unseen message count, and "next available UID". This operation is useful in synchronizing local and global mail states, since it allows a client to compare the user's global mailbox list with a client's local mailbox list. The list of mailboxes also provides a quick summary of each mailbox's contents without having the contents present.

The "create-mailbox" has the repository create a new mailbox and attach it to the user's list of mailboxes. An address object binding the (user-name, mailbox-name) pair to an RFC-822-style address is automatically created and placed in the repository's list of address objects. This allows mail coming from the Internet to be correctly routed to the new mailbox.

"Delete-mailbox" removes a mailbox from the user's list of mailboxes. All messages within the mailbox are also deleted and permanently removed from the system. Any address objects binding the mailbox name to RFC-822-style mailbox addresses are also removed from the system.

"Reset-mailbox" causes the repository to mark all the messages in a named mailbox as having changed since the current client last saw them. When the client next synchronizes with the repository, it will receive a complete copy of the named mailbox's mail state. This operation is merely a more specific version of the reset-client operation (which allows the client to pull over a complete copy of the user's global mail state). Its primary use is for mailboxes whose contents have accidentally been destroyed locally.

Finally, DMSP has an "expunge-mailbox" operation. Any message can be deleted and "undeleted" at will. Deletions are made permanent by performing an expunge-mailbox operation. The expunge operation causes the repository to look through a named mailbox, removing from the system any messages marked "deleted".

4.5. Address operations

DMSP provides three operations that allow users to manipulate address objects. First, the "list-address" operation returns a list of address objects associated with a particular (user-name, mailbox-name) pair.

The "create-address" operation adds a new address object that associates a (user-name, mailbox-name) pair with a given RFC-822-style mailbox address.

Finally, the "delete-address" operation destroys the address object binding the given RFC-822-style mail address and the given (user-name, mailbox-name) pair.

4.6. Bboard operations

DMSP provides seven bulletin board operations. The first, "list-bboards", gives the user a list of the bboards he is currently subscribing to. The list contains an entry for each bboard that the user subscribes to. Each entry contains the following information:

- The bulletin board's name
- The UID of the first message the subscriber has not yet seen
- The highest message UID in the bulletin board
- The number of messages the subscriber has not yet seen

"List-all-bboards" gives the user a list of all bboards he can subscribe to.

"Create-bboard" allows a user to create a bboard mailbox. The name given must be unique across the entire repository user community. Once the bboard mailbox has been created, other users may subscribe to the bboard, using bboard objects to keep track of which messages they have read on which bboards.

"Delete-bboard" allows a bboard's owner to delete a bboard mailbox. Subscribers who attempt to read from a bboard mailbox after it has been deleted are told that the bboard no longer exists.

DMSP also provides operations to subscribe to, and unsubscribe from, any bboard. "Subscribe-bboard" adds a bboard object to the users bboard object list; "unsubscribe-bboard" removes a bboard object from the list. Note that this does not delete the bboard mailbox (obviously only the bboard's owner can do that). It merely removes the user from the list of the bboard's subscribers.

DMSP allows for the user to tell the repository which messages in a bboard he has seen. Every bboard object contains the UID of the first message the user has not yet seen; the "set-first-unread-message-UID" operation updates that number, insuring that the user sees a given bboard message only once.

4.7. Message operations

The most commonly-manipulated Pcmail objects are messages; DMSP therefore provides special message operations to allow efficient synchronization, as well as a set of operations to perform standard message-manipulation functions. In the following paragraphs, the terms "message" and "descriptor" will be used interchangeably.

A user may request a series of descriptors with the "get-descriptors"

operation. The series is identified by a pair of message UIDs, representing the lower and upper bounds of the list. Since UIDs are defined to be monotonically increasing numbers, a pair of UIDs is sufficient to completely identify the series of descriptors. If the lower bound UID does not exist, the repository starts the series with the first message with UID greater than the lower bound. Similarly, if the upper bound does not exist, the repository ends the series with the last message with UID less than the upper bound. If certain UIDs within the series no longer exist, the repository (obviously) does not send them. The repository returns the descriptors in a sequence of "choices". Elements of the sequence can either be descriptors, in which case the choice is tagged as a descriptor, or they can be notification that the requested message has been expunged subsequent to the client's last connection to the repository. A descriptor choice is a record containing the message's UID, flags, to, from, date, and subject fields, length in bytes, and length in lines. An expunged choice contains only the expunged message's UID.

The "get-changed-descriptors" operation is intended for use during state synchronization. Whenever a descriptor changes state (is deleted, for example), the repository notes those clients which have not yet recorded the change locally. Get-changed-descriptors has the repository send to the client a given number of descriptors which have changed since the client's last synchronization. The list sent begins with the earliest-changed descriptor. Note that the list of descriptors is only guaranteed to be monotonically increasing for a given call to "get-changed-descriptors"; messages with lower UIDs may be changed by other clients in between calls to "get-changed-descriptors".

Once the changed descriptors have been looked at, a user will want to inform the repository that the current client has recorded the change locally. The "reset-changed-descriptors" causes the repository to mark as "seen by current client" a given series of descriptors. The series is identified by a low UID and a high UID. UIDs within the series that no longer exist are not reset.

Message bodies are transmitted from repository to user with the "get-message-text" operation. The separation of "get-descriptors" and "get-message-text" operations allows clients with small amounts of disk storage to obtain a small message summary (via "get-descriptors" or "get-changed-descriptors") without having to pull over the entire message.

Frequently, a message may be too large for some clients to store locally. Users can still look at the message contents via the "print-message" operation. This operation has the repository send a copy of the message to a named printer. The printer name need only have meaning to the particular repository implementation; DMSP transmits the name only as a means of identification.

Copying of one message into another mailbox is accomplished via the "copy-message" operation. A descriptor list of length one, containing a descriptor for the copied message is returned if the copy operation is successful. This descriptor is required because the copied message acquires a UID different from the original message. The client cannot be expected to know which UID has been assigned the copy, hence the repository's sending a descriptor containing the UID.

5. Client Architecture

Clients can be any of a number of different workstations; Pcmail's architecture must therefore take into account the range of characteristics of these workstations. First, most workstations are much more affordable than the large computers currently used for mail service. It is therefore possible that a user may well have more than one. Second, some workstations are portable and they are not expected to be constantly tied into a network. Finally, many of the smaller workstations resource-poor, so they are not expected to be able to store a significant amount of state information locally. The following subsections describe the particular parts of Pcmail's client architecture that address these different characteristics.

5.1. Multiple clients

The fact that Pcmail users may own more than one workstation forms the rationalization for the multiple client model that Pcmail uses. A Pcmail user may have one client at home, another at an office, and maybe even a third portable client. Each client maintains a separate copy of the user's mail state, hence Pcmail's distributed nature. The notion of separate clients allows Pcmail users to access mail state from several different locations. Pcmail places no restrictions on a user's ability to communicate with the repository from several clients at the same time. Instead, the decision to allow several clients concurrent access to a user's mail state is made by the repository implementation.

5.2. Synchronization

Some workstations tend to be small and fairly portable; the likelihood of their always being connected to a network is relatively small. This is another reason for each client's maintaining a local copy of a user's mail state. The user can then manipulate the local mail state while not connected to the network (and the repository). This immediately brings up the problem of synchronization between local and global mail states. The repository is continually in a position to receive global mail state updates, either in the form of incoming mail, or in the form of changes from other clients. A client that is not always connected to the net cannot immediately receive the global changes. In addition, the client's user can make his own changes on the local mail state.

Pcmail's architecture allows fast synchronization between client local mail states and the repository's global mail state. Each client is identified in the repository by a client object attached to the user. This object forms the basis for synchronization between local and global mail states. Some of the less common state changes include the adding and deleting of user mailboxes and the adding and deleting of address objects. Synchronization of these changes is performed via DMSP list operations, which allow clients to compare their local versions of mailbox and address object lists with the repository's global version and make any appropriate changes. The majority of possible changes to a user's mail state are in the form of changed descriptors. Since most users will have a large number of messages, and message states will change relatively often, special attention needs to be paid to message synchronization.

An existing descriptor can be changed in one of two ways: first, one of its sixteen flags values can be changed (this encompasses reading an unseen message, deleting a message, and expunging a message). The second way to change a descriptor is via the arrival of incoming mail or the copying of a message from one mailbox to another. Both result in a new message being added to a mailbox.

In both the above cases, synchronization is required between the repository and every client that has not previously noted a change. To keep track of which clients have noticed a global mail state change and changed their local states accordingly, each mailbox has associated with it a list of active clients. Each client has a (potentially empty) "update list" of messages which have changed since that client last read them.

When a client connects to the repository, it executes a DMSP "get-changed-descriptors" operation. This causes the repository to return a list of all descriptor objects on that client's update list As the client receives the changed descriptors, it can store them locally, thus updating the local mail state. After a changed descriptor has been recorded, the client uses the DMSP "reset-descriptors" operation to remove the message from its update list. That descriptor will now not be sent to the client unless (1) it is explicitly requested, or (2) it changes again.

In this manner, a client can run through its user's mailboxes, getting all changed descriptors, incorporating them into the local mail state, and marking the change as recorded.

5.3. Batch operation versus interactive operation

Because of the portable nature of some workstations, they may not always be connected to a network (and able to communicate with the repository). Since each client maintains a local mail state, Pcmail users can manipulate the local state while not connected to the repository. This is known as "batch" operation, since all changes are

recorded by the client and made to the repository's global state in a batch, when the client next connects to the repository. Interactive operation occurs when a client is always connected to the repository. In interactive mode, changes made to the local mail state are also immediately made to the global state via DMSP operations.

In batch mode, interaction between client and repository takes the following form: the client connects to the repository and sends over all the changes made by the user to the local mail state. The repository changes its global mail state accordingly. When all changes have been processed, the client begins synchronization, to incorporate newly-arrived mail, as well as mail state changes by other clients, into the local state.

In interactive mode, since local changes are immediately propagated to the repository, the first part of batch-type operation is eliminated. The synchronization process also changes; although one synchronization is required when the client first opens a connection to the repository, subsequent synchronizations can be performed either at the user's request or automatically every so often by the client.

5.4. Message summaries

Smaller workstations may have little in the way of disk storage. Clients running on these workstations may never have enough room for a complete local copy of a user's global mail state. This means that Pcmail's client architecture must allow user's to obtain a clear picture of their mail state without having all their messages present.

Descriptors provide message information without taking up large amounts of storage. Each descriptor contains a summary of information on a message. This information includes the message UID, its length in bytes and lines, its status (encoded in the eight system-defined and eight user-defined flags), and portions of its RFC-822 header (the "to:", "from:", "subject:" and "date:" fields). All of this information can be encoded in a small (around 100 bytes) data structure whose length is independent of the size of the message it describes.

Most clients should be able to store a complete list of message descriptors with little problem. This allows a user to get a complete picture of his mail state without having all his messages present locally. If a client has extremely limited amounts of disk storage, it is also possible to get a subset of the descriptors from the repository. Short messages can reside on the client, along with the descriptors, and long messages can either be printed via the DMSP print-message operation, or specially pulled over via the fetch-message-text operation.

6. Typical interactive-style client-repository interaction

The following example describes a typical communication session between the repository and a client. The client is one of three belonging to user "Fred". Its name is "office-client", and since Fred uses the client regularly to access his mail, the client is marked as "active". Fred has two mailboxes: "main" is where all of his current mail is stored; "archive" is where messages of lasting importance are kept. The example will run through a simple synchronization operation followed by a series of typical mail state manipulations. Typically, the synchronization will be performed by an application program that connects to the repository, logs in, synchronizes, and logs out.

For the example, all DMSP operations will be shown in a user-readable format. In reality, the operations would be sent as a stream of USP blocks consisting of a block-type number followed by a stream of bytes representing the block's components.

In order to access his global mail state, the client software must authenticate Fred to the repository; this is done via the DMSP login operation:

```
login ["fred", "fred-password", "office-client", F, F]
```

This tells the repository that Fred is logging in via "office-client", and that "office-client" is identified by an existing client object attached to Fred's user object. The first component of the login block is Fred's repository user name. The second component is Fred's password. The third component is the name of the client communicating with the repository. The fourth component tells the repository not to create "office-client" even if it cannot find its client object. The final component tells the repository that Fred's client is not operating in batch mode but rather in interactive mode.

Fred's authentication checks out, so the repository logs him in, acknowledging the login request with an OK block.

Now that Fred is logged in, the client performs an initial synchronization. This process starts with the client's asking for an up-to-date list of mailboxes:

```
list-mailboxes []
```

The repository replies with:

```
mailbox-list [["main", 10, 1, 253],
              ["archive", 100, 0, 101]]
```

This tells the client that there are two mailboxes, "main" and "archive". "Main" has 10 messages, one of which is unseen. The next

incoming message will be assigned a UID of 253. "Archive", on the other hand, has 100 message, none of which are unseen. The next message sent to "archive" will be assigned the UID 101. There are no new mailboxes in the list (if there were, the client program would create them. On the other hand, if some mailboxes in the client's local list were not in the repository's list, the program would assume them deleted by another client and delete them locally as well).

To synchronize the client need only look at each mailbox's contents to see if (1) any new mail has arrived, or (2) if Fred changed any messages on one of his other two clients subsequent to "office-client"'s last connection to the repository.

The client asks for any changed descriptors via the "get-changed-descriptors" operation. It requests at most ten changed descriptors since storage is very limited on "office-client".

```
get-changed-descriptors ["main", 10]
```

The repository responds with:

```
descriptor-list [[descriptor[
                        6,
                        [T T F F F F F F F F F F
                         F F F F],
                        "Fred@borax",
                        "Joe@fab",
                        "Wed, 23 Jan 86 11:11 EST",
                        "tomorrow's meeting",
                        621,
                        10]]
                [descriptor[
                        10,
                        [F T F F F F F F F F F F
                         F F F F],
                        "Fred",
                        "Freds-secretary",
                        "Fri, 25 Jan 86 11:11 EST",
                        "Monthly progress report",
                        13211,
                        350]]
                ]
```

The first descriptor in the list is one which Fred deleted on another client yesterday. "Office-client" marks the local version of the message as deleted. The second descriptor in the list is a new one. "Office-client" adds the descriptor to its local list. Since both changes have now been recorded locally, the descriptors can be reset:

```
reset-descriptors ["main", 6, 10]
```

The repository removes from "office-client"'s update list all messages with UIDs between 6 and 10 (in this case just two messages) "Main" has now been synchronized. The client now turns to Fred's "archive" mailbox and asks for the first ten changed descriptors.

```
get-changed-descriptors ["archive", 10]
```

The repository responds with

```
descriptor-list []
```

The zero-length list tells "office-client" that no descriptors have been changed in "archive" since its last synchronization. No new synchronization needs to be performed.

Fred's client is now ready to pull over the new message. The message is 320 lines long; there might not be sufficient storage on "office-client" to hold the new message. The client tries anyway:

```
fetch-message-text ["main", 10]
```

The repository begins transmitting the message:

```
message ["From: Fred's-secretary",
         "To: Fred",
         "Subject: Monthly progress report",
         "Date: Fri, 25 Jan 86 11:11 EST",
         "",
         "Dear Fred,",
         "Here is this month's progress report",
         ...
         ]
```

Halfway through the message transmission, "office-client" runs out of disk space. Because all DMSP operations are defined to be failure-atomic, the portion of the message already transmitted is destroyed locally and the operation fails. "Office-client" informs Fred that the message cannot be pulled over because of a lack of disk space. The synchronization process is now finished and Fred can start reading his mail. The new message that was too big to fit on "office-client" will be marked "off line"; Fred can either remote-print it or delete other messages until he has enough space to store the new message.

Since he is running in interactive mode, changes he makes to any messages will immediately be transmitted into DMSP operations and sent to the repository. Depending on the client implementation, Fred will either have to execute a "synchronize" command periodically or the client will synchronize for him automatically every so often.

7. A current Pcmail implementation

The following section briefly describes a current Pcmail system that services a small community of users. The Pcmail repository runs under UNIX on a DEC VAX-750 connected to the Internet. The clients run on IBM PCs, XTs, and ATs, as well as Sun workstations, Microvaxes, and VAX-750s.

7.1. IBM PC client code

Client code for the IBM machines operates only in batch mode. Users make local state changes, which are queued until the client connects to the repository. At that time, the changes are performed and the local and global states synchronized. The client then disconnects from the repository.

Users access and modify their local mail state via a user interface program. The program uses windows and a full-screen mode of operation. Users are given a variety of commands to operate on individual messages as well as mailboxes. The interface allows use of any text editor to compose messages, and adds features of its own to make RFC-822-style header composition easier.

Synchronization and the processing of queued changes is performed by a separate program, which the user runs whenever he wishes. The program takes any actions queued while operating the user interface, and converts them into DMSP operations. All queued changes are made before any synchronization is performed.

The limitation of IBM PC client operation to batch mode was made because of development environment limitations. The user interface could not work with the network code inside it due to program size limitations. Since MS-DOS has no multi-processing facilities, the two programs could not run in tandem either. The only solution was to provide a two-part client, one part of which read the mail and one part of which interacted with the repository.

7.2. UNIX client code

Client code for the Suns, Microvaxes, and VAX-750s runs on 4.2/4.3BSD UNIX. It is fully interactive, with a powerful user interface inside Richard Stallman's GNU-EMACS editor. Since UNIX-based workstations have a good deal of main memory and disk storage, no effort was made to lower local mail state size by keeping message descriptors rather than message text.

The local mail state consists of a number of BABYL-format mailboxes. The interface is very similar to the RMAIL mail reader already present in GNU-EMACS.

The user interface communicates with the repository through a DMSP

implementation built into the GNU-EMACS kernel. Changes to the local mail state are immediately made on the repository; the repository is fast enough that there is little noticeable delay in performing the operation over the network.

There is no provision for automatic synchronization whenever new mail arrives or old mail is changed by another client. Instead, users must get any new mail explicitly. A simple "notification" program runs in the background and wakes up every minute to check for new mail; when mail arrives, the user executes a command to get the new mail, synchronizing the mailbox at the same time.

7.3. Repository code

The repository is implemented in C on 4.2/4.3BSD UNIX. Currently it runs on DEC VAX-750s and Microvaxes, although other repositories will soon be running on IBM RT machines and Sun workstations. The repository code is designed to allow several clients belonging to a particular user to "concurrently" modify the user's state. A mailbox locking scheme prevents one client from modifying a mailbox while another client is modifying the same mailbox.

8. Conclusions

Pcmail is now used by a small community of people at the MIT Laboratory for Computer Science. The repository design works well, providing an efficient means of storing and maintaining mail state for several users. Its performance is quite good when up to ten users are connected; it remains to be seen whether or not the repository will be efficient at managing the state of ten or a hundred times that many users. Given sufficient disk storage, it should be able to, since communication between different users' clients and the repository is likely to be very asynchronous and likely to occur in short bursts with long "quiet intervals" in between as users are busy doing other things.

Members of another research group at LCS are currently working on a replicated, scalable version of the repository designed to support a very large community of users with high availability. This repository also uses DMSP and has successfully communicated with clients that use the current repository implementation. DMSP therefore seems to be usable over several flavors of repository design.

The IBM PC clients are unfortunately very limited in the way of resources, making local mail state manipulation difficult at times. Synchronization is also relatively time consuming due to the low performance of the PCs. The "batch-mode" that the PCs use tends to be good for those PCs that spend a large percentage of their time unplugged and away from a network. It is somewhat inconvenient for those PCs that are always connected to a network and could make good use of an "interactive-mode" state manipulation.

The UNIX-based clients are far easier to use than their PC counterparts. Synchronization is much faster, and there is far more functionality in the user interface (having an interface that runs within GNU-EMACS helps a lot in this respect). Most of those people using the Pcmail system use the UNIX-based client code.

APPENDIX

A. DMSP Protocol Specification

Following are a list of DMSP operations by object type, their block types and arguments, and their expected acknowledgement block types. Each DMSP block has a different number; the first digit of each block type defines the object being manipulated: Operations numbered 5xx are general, operations numbered 6xx are user operations, operations numbered 7xx are client operations, operations numbered 8xx are mailbox operations, operations numbered 9xx are address operations, operations numbered 10xx are bboard operations, and operations numbered 11xx are message operations.

Failure blocks contain two fields, a "code" and a "why". The "code" is an unsigned number placing the error in one of several broad categories (listed below). The "why" is a text string, possibly explaining the error in greater detail.

Error codes:

- 1: network error while reading or writing data

- 2: internal repository error. This can be due to lack of memory, a fatal bug, lack of disk space, etc.

- 3: requested object already exists. For example, you tried to create a mailbox that already exists

- 4: requested object not found. For example, you tried to delete a message or a mailbox that doesn't exist.

- 5: protocol error. Typically DMSP protocol version skew.

- 6: block argument error. For example, a "set-message-flag" operation was attempted on a bboard by someone other than the bboard's owner.

- 7: data read error. The repository was unable to read the mail state information requested.

- 8: data write error. The repository was unable to write out changed mail state information, perhaps because the disk was full.

- 9: operating system error: Should be reserved for things like fork or pipe call errors.

- 10: unexpected or unknown block type received. For example, you sent a "delete-mailbox" block and received a "mailbox-list" block in response.

Blocks marked "=>" flow from client to repository; blocks marked "<=" flow from repository to client. If more than one block can be sent, the choices are delimited by "or" ("|") characters.

For clarity, each block type is put in a human-understandable form. The block number is followed by an operation name; this name is never transmitted as part of a USP block. Block arguments are identified by name and type, and enclosed in square brackets. "Record" data types are described by a list of "field-name:field-type" pairs contained in square brackets. "Choice" data types are described by a list of "tag-name:tag-type" pairs contained in square brackets. USP data types are defined as follows (the definitions are brief; refer to the USP specification for more detailed descriptions):

A.1. Primitive data types

string (S): a series of bytes, null-byte padded to even length and preceded by a 16-bit length specifier. Strings are sent in "net-ascii" format (newline sequence is carriage return followed by linefeed, single carriage returns to be followed by a null byte).

- cardinal (C): a 16-bit unsigned number.

- long-cardinal (LC): a 32-bit unsigned number.

- integer (I): a 16-bit signed number.

- long-integer (LI): a 32-bit signed number.

- boolean (B): a 16-bit number with either a 1 or a 0 in the 16th bit.

A.2. Compound data types

- sequence (SEQ): A list of data items, all the same type and preceded by a 16-bit sequence length specifier.

- array (AR): A fixed-length list of data items, all the same type. A particular array's length is fixed by the application.

- record (REC): A list of data items of any type. A particular record's format is fixed by the application.

- choice (CH): One of a list of possible data types. The data type contained in the choice is identified by a 16-bit numeric tag. The application interprets the data item based on the tag value.

A.3. DMSP Abstract Data Types

Following are data types defined and used only by DMSP:

- client: a record with the following format:
  REC[name:S, status:C] Status is either 1 (active) or 0 (inactive)

- mailbox: a record with the following format:
  REC[name:S, next-uid:LC, #msgs:C, #new-msgs:C]

- bboard: a record with the following format:
  REC[name:S, first-unread-message-UID:LC
  number-of-unseen-messages:C highest-UID:LC]

- descriptor: a record with the following format:
  - REC[UID:LC, flags:SEQ[B], from, to, date, subject:S, #bytes:LC, #lines:LC]

- desc-choice: a choice with the following format:
  CH[expunged-message-UID:LC, desc:descriptor] Descriptor tag number is 1. Expunged-message tag number is 0.

A.4. General operations

```
=> 502 (send-version) [version:C]
<= 500 (ok) [] |
   501 (failure) [code:C, why:S]


=> 503 (send-message) [message:SEQ[S]]
<= 500 (ok) [] |
   501 (failure) [code:C, why:S]
```

A.5. User operations

```
     => 600 (login) [name:S, password:S, client:S,
                     create-client-object?:B
                     batch-mode?:B]
     <= 500 (ok) [] |
        501 (failure) [code:C, why:S] |
        705 (force-client-reset) []


     => 601 (logout) []
     <= 500 (ok) []


     => 602 (set-password) [old:S, new:S]
     <= 500 (ok) [] |
        501 (failure) [code:C, why:S]
```

A.6. Client operations

```
     => 701 (list-clients) []
     <= 700 (client-list) [client-list:SEQ[client]]


     => 702 (create-client) [client:S]
     <= 500 (ok) [] |
        501 (failure) [code:C, why:S]


     => 703 (delete-client) [client:S]
     <= 500 (ok) [] |
        501 (failure) [code:C, why:S]


     => 704 (reset-client) [client:S]
     <= 500 (ok) [] |
        501 (failure) [code:C, why:S]
```

A.7. Mailbox operations

```
     => 801 (list-mailboxes) []
     <= 800 (mailbox-list) [mailbox-list:SEQ[mailbox]]


     => 802 (create-mailbox) [mailbox:S]
     <= 500 (ok) [] |
        501 (failure) [code:C, why:S]


     => 803 (delete-mailbox) [mailbox:S]
     <= 500 (ok) [] |
        501 (failure) [code:C, why:S]
```

```
=> 804 (reset-mailbox) [mailbox:S]
<= 500 (ok) [] |
   501 (failure) [code:C, why:S]


=> 805 (expunge-mailbox) [mailbox:S]
<= 500 (ok) [] |
   501 (failure) [code:C, why:S
```

A.8. Address operations

```
=> 901 (list-addresses) [mailbox:S]
<= 501 (failure) [code:C, why:S] |
   900 (address-list) [address-list:SEQ[S]]


=> 902 (create-address) [mailbox:S, address:S]
<= 500 (ok) [] |
   501 (failure) [code:C, why:S]


=> 903 (delete-address) [mailbox:S, address:S]
<= 500 (ok) [] |
   501 (failure) [code:C, why:S]
```

A.9. Bboard operations

```
=> 1001 (list-bboards) []
<= 1000 (bboard-list) [bboard-list:SEQ[bboard]]
   501 (failure) [code:C, why:S]


=> 1002 (create-bboard) [name:S]
<= 500 (ok) [] |
   501 (failure) [code:C, why:S]


=> 1003 (delete-bboard) [name:S]
<= 500 (ok) [] |
   501 (failure) [code:C, why:S]


=> 1004 (subscribe-bboard) [name:S]
<= 500 (ok) [] |
   501 (failure) [code:C, why:S]


=> 1005 (unsubscribe-bboard) [name:S]
<= 500 (ok) [] |
   501 (failure) [code:C, why:S]
```

```
     => 1006 (set-bboard-first-unread) [name:S, UID:LC]
     <= 500 (ok) [] |
        501 (failure) [code:C, why:S]

     => 1007 (list-all-bboards) []
     <= 1008 (bboard-name-list) [bboard-name-list:SEQ[S]]
        501 (failure) [code:C, why:S]
```

A.10. Message operations

```
     => 1102 (get-descriptors) [mailbox:S,
                                low-uid:LC,
                                high-uid:LC]
     <= 501 (failure) [code:C, why:S] |
        1100 (desc-list) [desc-list:SEQ[desc-choice]]


     => 1103 (get-changed-descriptors) [mailbox:S,
                                        max-to-send:C]
     <= 501 (failure) [code:C, why:S] |
        1100 (desc-list) [desc-list:SEQ[desc-choice]]


     => 1104 (reset-changed-descriptors) [
                    mailbox:S,
                    start-uid:LC,
                    end-uid:LC]
     <= 500 (ok) [] |
        501 (failure) [code:C, why:S]


     => 1105 (get-message-text) [mailbox:S,
                                 uid:LC]
     <= 501 (failure) [code:C, why:S] |
        1101 (message) [message:SEQ[S]]


     => 1106 (print-message) [mailbox:S,
                              uid:LC,
                              printer-name:S]
     <= 500 (ok) [] |
        501 (failure) [code:C, why:S]


     => 1107 copy-message[source-mailbox:S,
                          target-mailbox:S,
                          source-uid:LC]
     <= 501 (failure) [code:C, why:S]
     <= 501 (failure) [code:C, why:S] |
        1100 (desc-list) [desc-list:SEQ[desc-choice]]
```

```
=> 1108 (set-flag) [mailbox:S,
                    uid:LC,
                    flag-number:C,
                    flag-setting:B]
<= 500 (ok) [] |
   501 (failure) [code:C, why:S]                                    30
```

DMSP block types by number

General block types

ok	500
failure	501
send-version	502
send-message	503

User operation block types

login	600
logout	601
set-password	602

Client operation block types

client-list	700
list-clients	701
create-client	702
delete-client	703
reset-client	704
force-client-reset	705

Mailbox operation block types

mailbox-list	800
list-mailboxes	801
create-mailbox	802
delete-mailbox	803
reset-mailbox	804
expunge-mailbox	805

Address operation block types

address-list	900
list-addresses	901
create-address	902
delete-address	903

Bboard operation block types

bboard-list	1000
list-bboards	1001
create-bboard	1002
delete-bboard	1003
subscribe-bboard	1004
unsubscribe-bboard	1005
set-bboard-first-unread	1006
get-n-new-bboard-descriptors	1007
list-all-bboards	1008
bboard-name-list	1009

Message operation block types

descriptor-list	1100
message	1101
get-descriptors	1102
get-changed-descriptors	1103
reset-changed-descriptors	1104
get-message-text	1105
print-message	1106
copy-message	1107
set-flag	1108